

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО  
ОБРАЗОВАНИЯ**  
**«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ  
УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И  
ОПТИКИ»**

**Факультет безопасности информационных технологий**  
*Кафедра проектирования и безопасности компьютерных систем*

**Дисциплина:**  
**«Операционные системы»**

**ОТЧЕТ О ВЫПОЛНЕНИИ  
ЛАБОРАТОРНОЙ РАБОТЫ №3**

**Выполнил:**  
Студент гр. N3247 Гаврилова В. В.

**Проверил:**  
Ханов А. Р.

Санкт-Петербург  
2021г.

Задание:

Часть 1

Найти и скомпилировать программу linpack для оценки производительности компьютера (Flops) и протестировать ее при различных режимах работы ОС:

- С различными приоритетами задачи в планировщике
- С наличием и отсутствием привязки к процессору
- Провести несколько тестов, сравнить результаты по 3 сигма или другим статистическим критериям

Часть 2

- Запретить выполнение всех потоков кроме того, который тестируется (путем запрета прерываний или найти иной способ (найден))

Выполнение:

**Часть 1 (Parrot OS)**

- 1) Запуск linpack с различными приоритетами: -20; -5; 0; 5; 19

Пример для 19:

```
out=$( nice -n 19 ./linpack | grep 65536 )
```

Вывод:

```
[root@parrot]# ./test3
17.57, 6758280.090
17.65, 6717953.426
17.55, 6771711.230
17.61, 6732069.952
17.55, 6767575.120
17.57, 6745080.558
17.59, 6746015.857
17.69, 6710147.080
17.71, 6695890.103
17.47, 6795842.080
17.63, 6727807.094
17.66, 6715933.235
17.71, 6684178.675
17.54, 6770058.319
17.50, 6789005.298
```

- 2) Запуск с привязкой к ядру 1:

```
out=$( taskset -c 1 chrt -f 1 ./linpack | grep 65536 )
```

- 3) Запуск с привязкой к ядру 1 и приоритетом 19:

```
out=$( nice -n 19 taskset -c 1 chrt -f 1 ./linpack | grep 65536 )
```

4) Анализ полученных результатов:

Для taskset -c 1 chrt -f 1

Дисперсия D для time: 0,015485

Дисперсия D для flops: 2921985828

Математическое ожидание m для time: 17,735

Математическое ожидание m для flops: 6680464,699

График плотности вероятности для time:

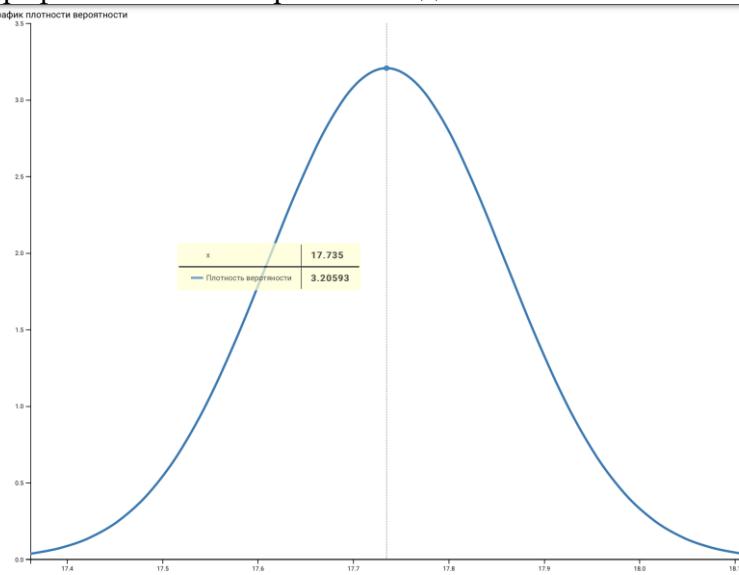


График функции распределения для time:

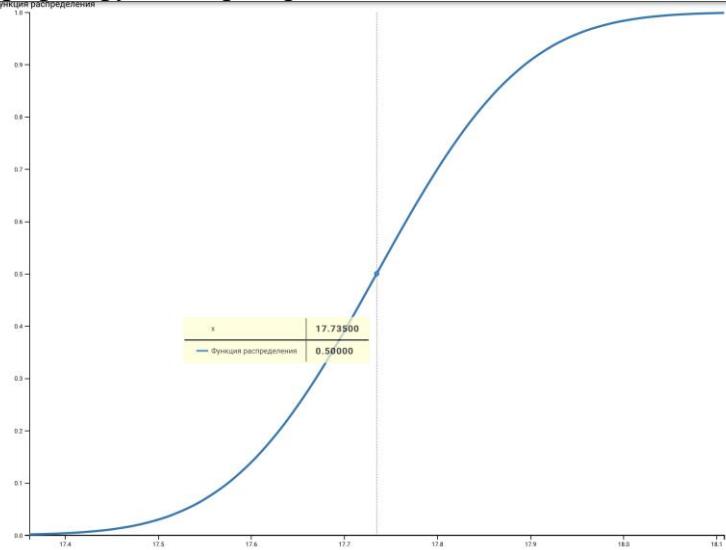


График плотности вероятности для flops:

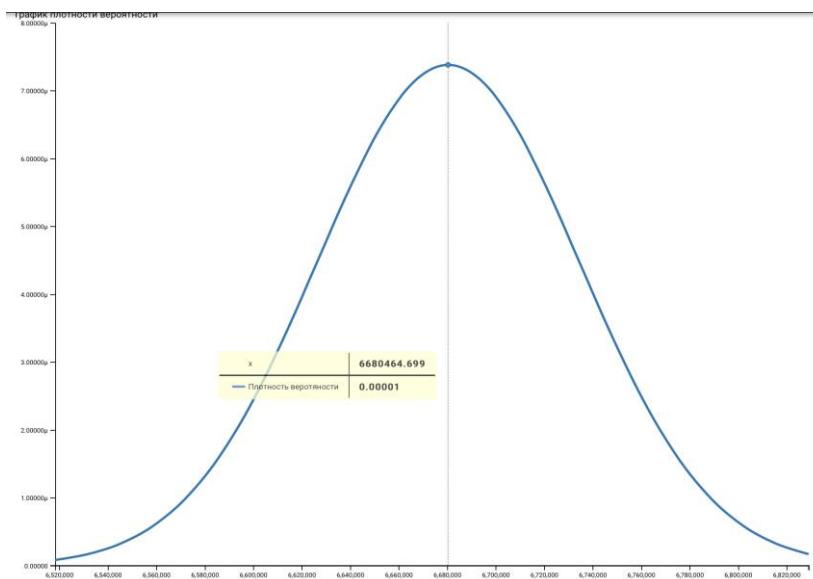
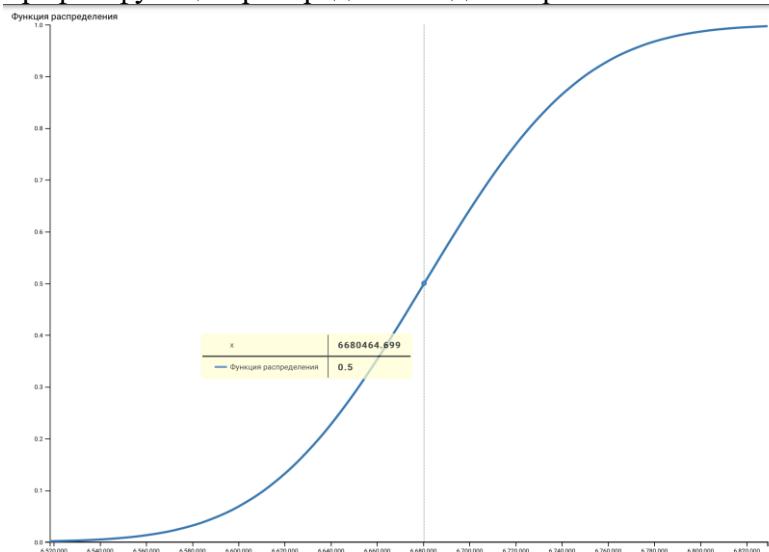


График функции распределения для flops:



Для nice -n -20

Дисперсия D для time: 0,020996

Дисперсия D для flops: 3925920847

Математическое ожидание m для time: 17,678

Математическое ожидание m для flops: 6705244,655

График плотности вероятности для time:

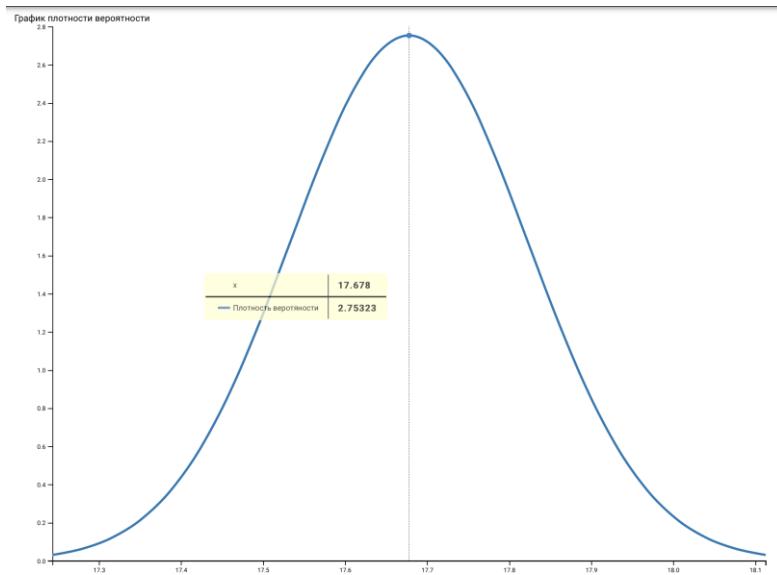


График функции распределения для time:

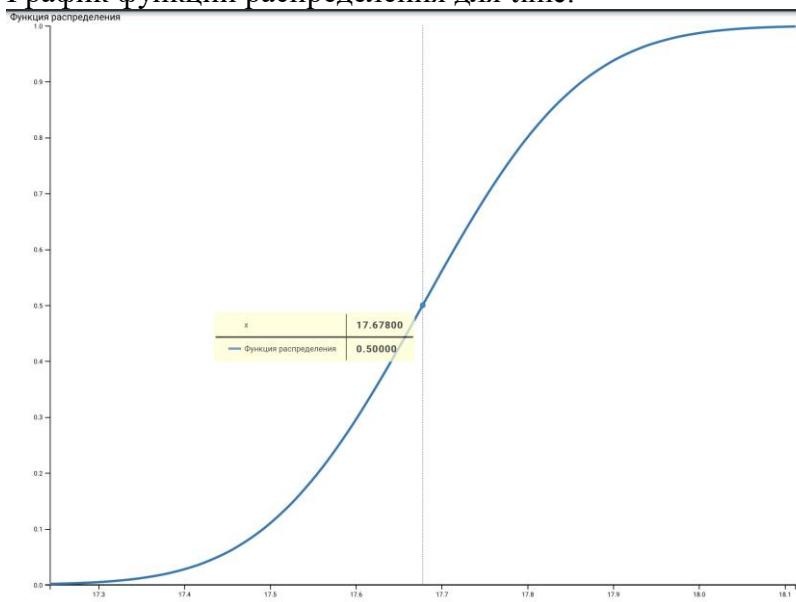


График плотности вероятности для flops:

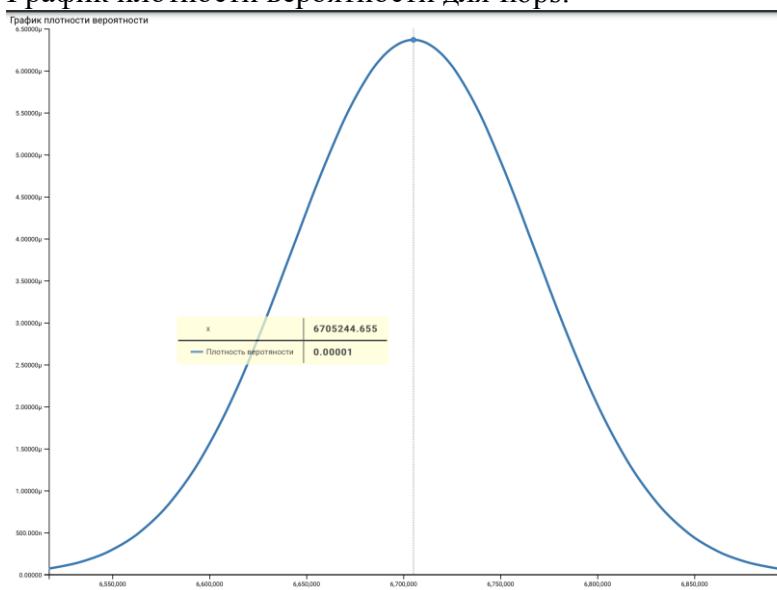
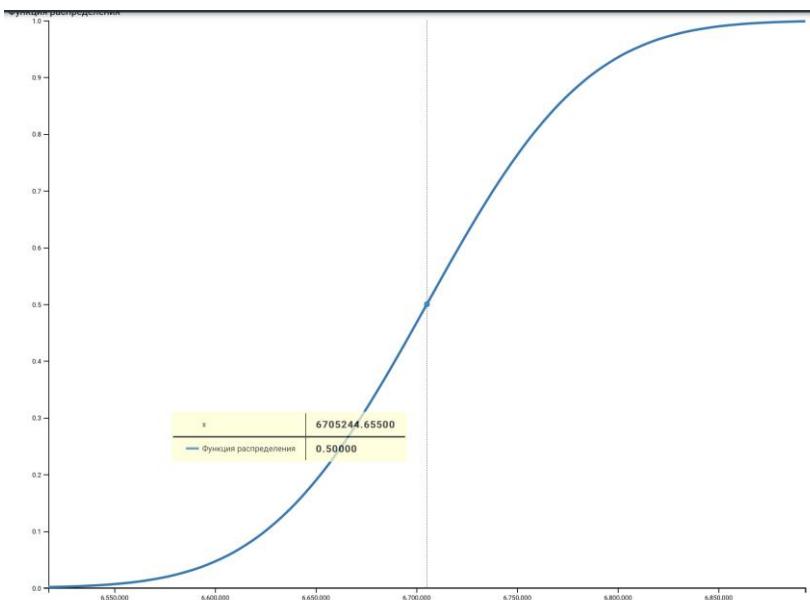


График функции распределения для flops:



Для nice -n -5

Дисперсия D для time: 0,04996267

Дисперсия D для flops: 7810439252

Математическое ожидание m для time: 17,718

Математическое ожидание m для flops: 6693279,182

График плотности вероятности для time:

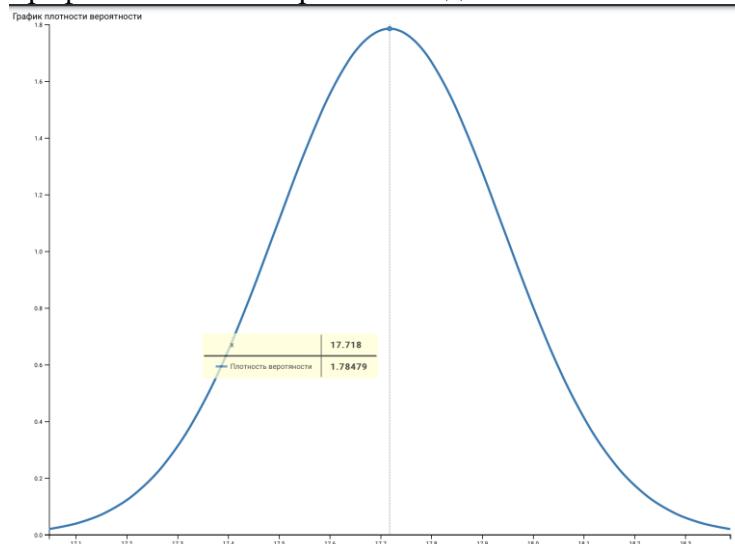


График функции распределения для time:

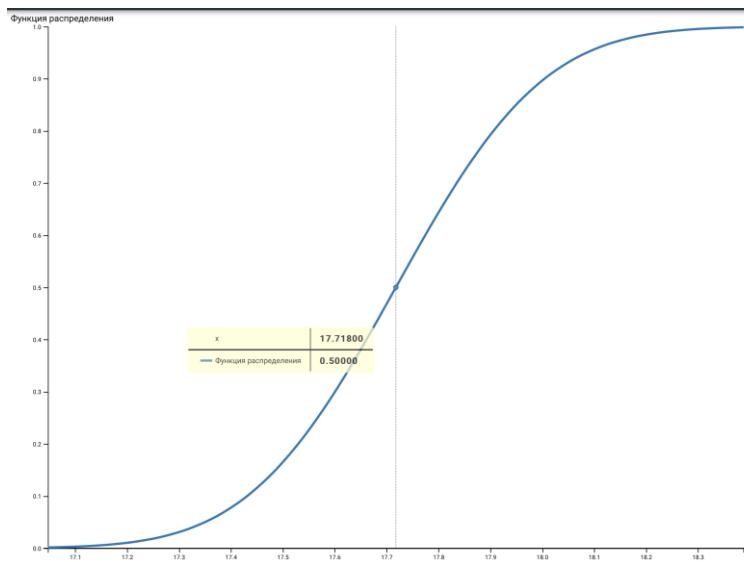


График плотности вероятности для flops:

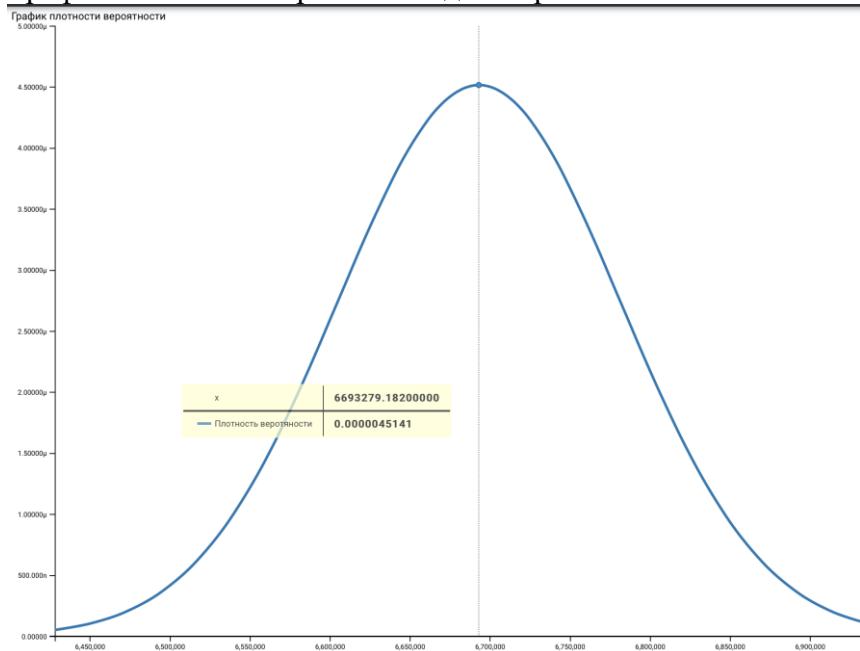
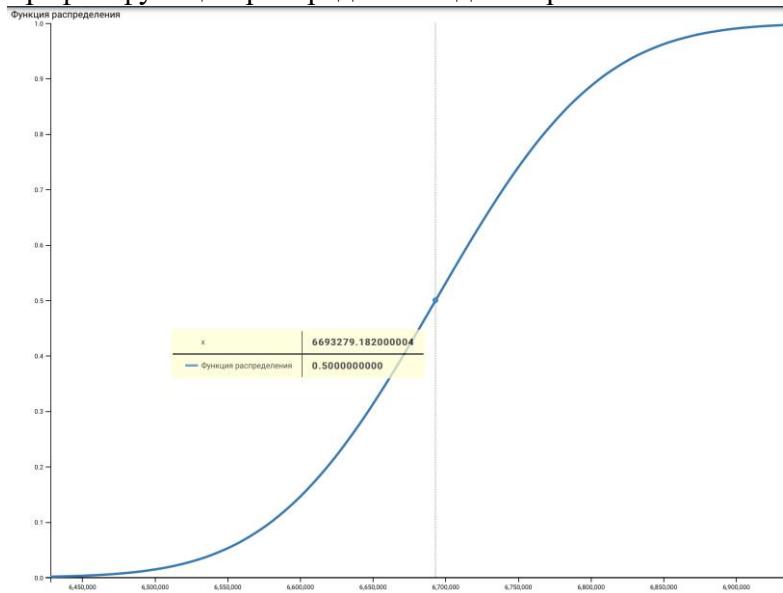


График функции распределения для flops:



**Для nice -n 0**

Дисперсия D для time: 0,059904

Дисперсия D для flops: 9494506357

Математическое ожидание m для time: 17,734

Математическое ожидание m для flops: 6688874,329

График плотности вероятности для time:

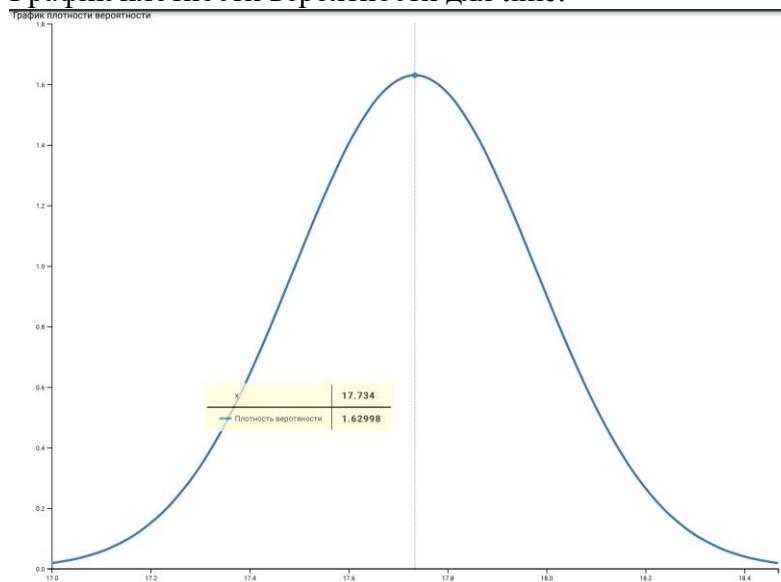


График функции распределения для time:

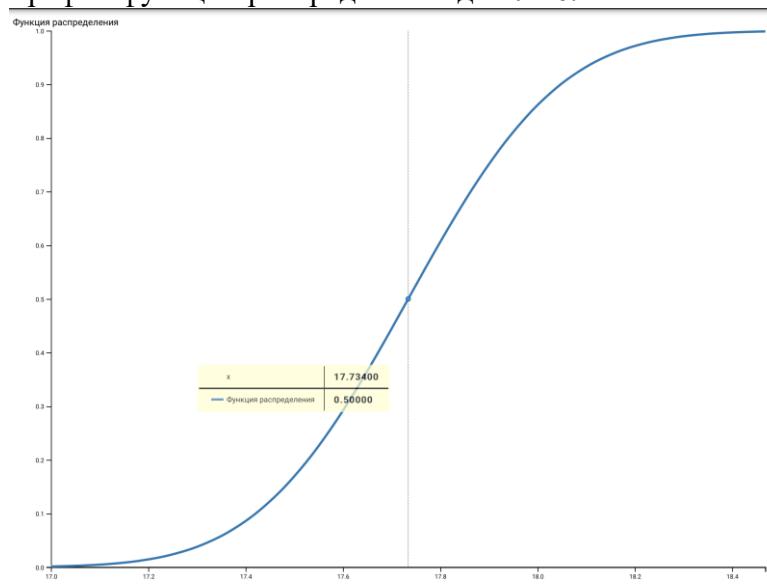


График плотности вероятности для flops:

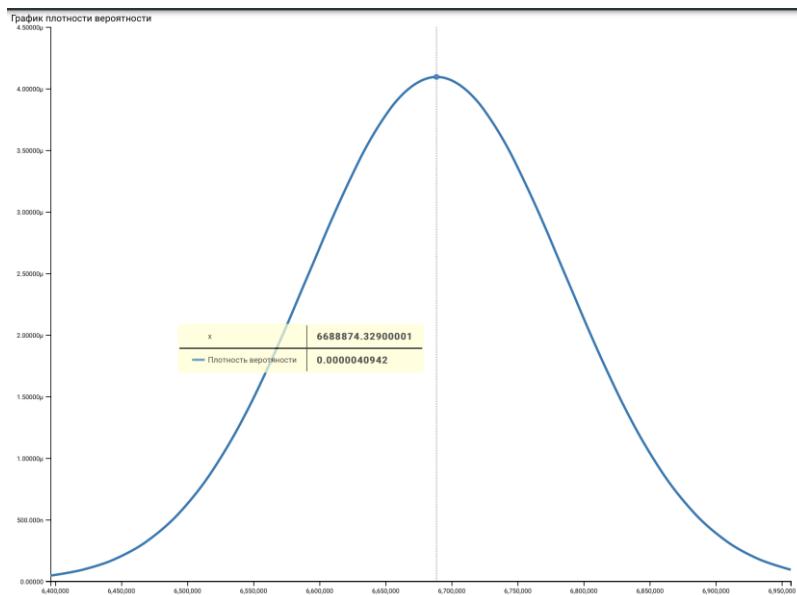
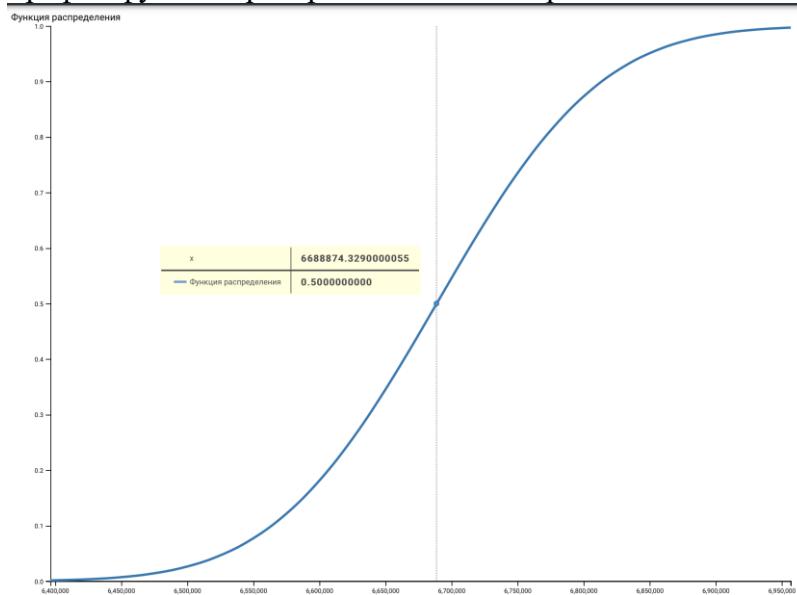


График функции распределения для flops:



Для nice -n 5

Дисперсия D для time: 0,09191733

Дисперсия D для flops: 14526646082

Математическое ожидание m для time: 17,796

Математическое ожидание m для flops: 6662014,95

График плотности вероятности для time:

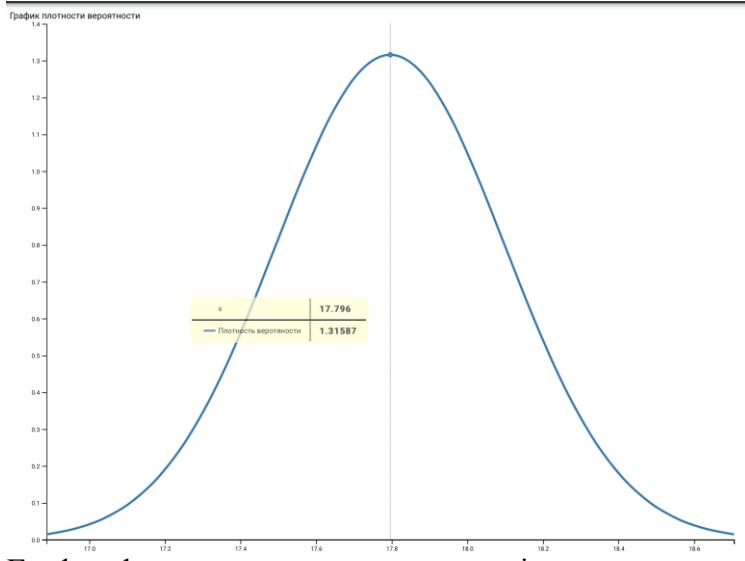


График функции распределения для time:

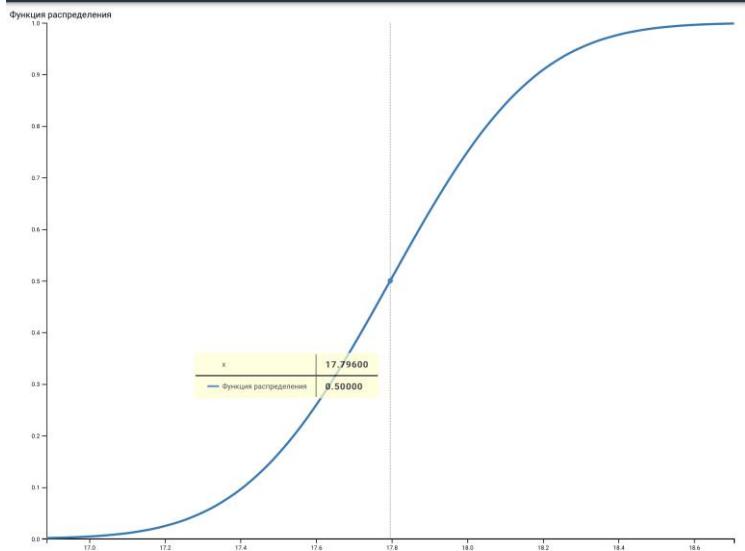


График плотности вероятности для flops:

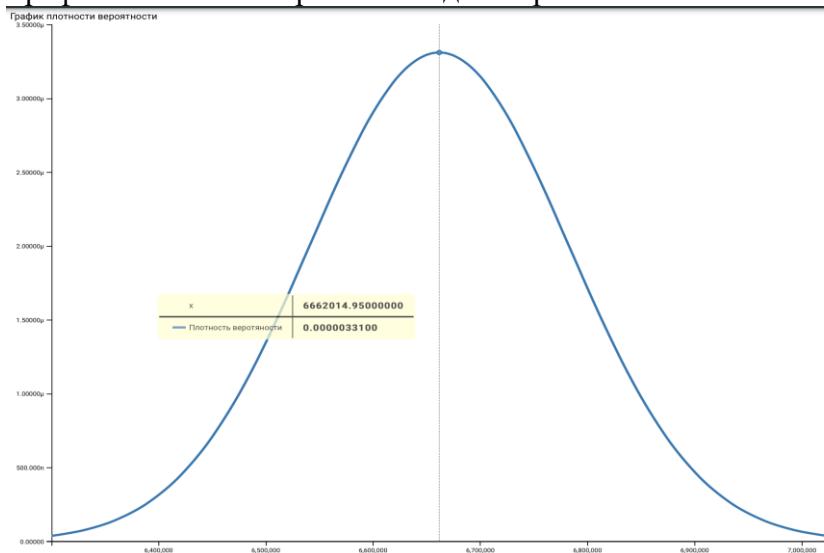
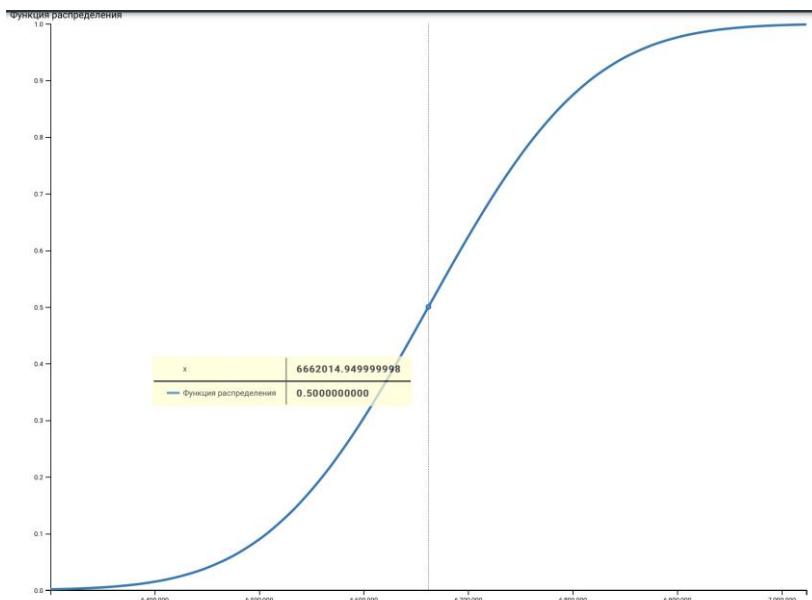


График функции распределения для flops:



Для nice -n 19

Дисперсия D для time: 0,03507733

Дисперсия D для flops: 6225004994

Математическое ожидание m для time: 17,744

Математическое ожидание m для flops: 6678302,26

График плотности вероятности для time:

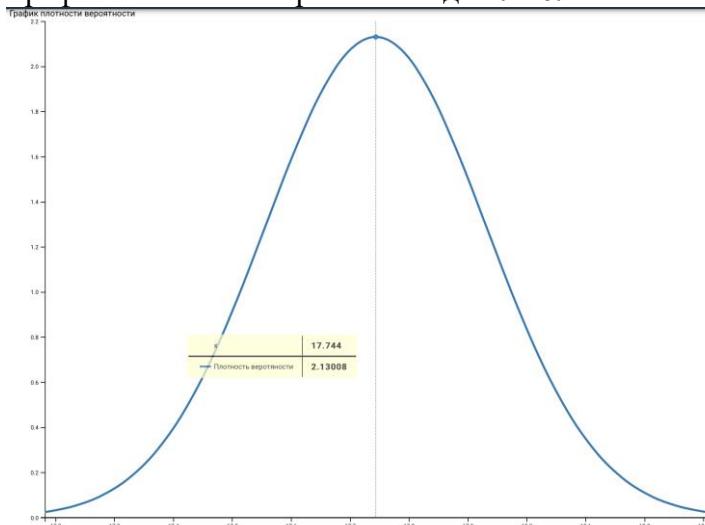


График функции распределения для time:

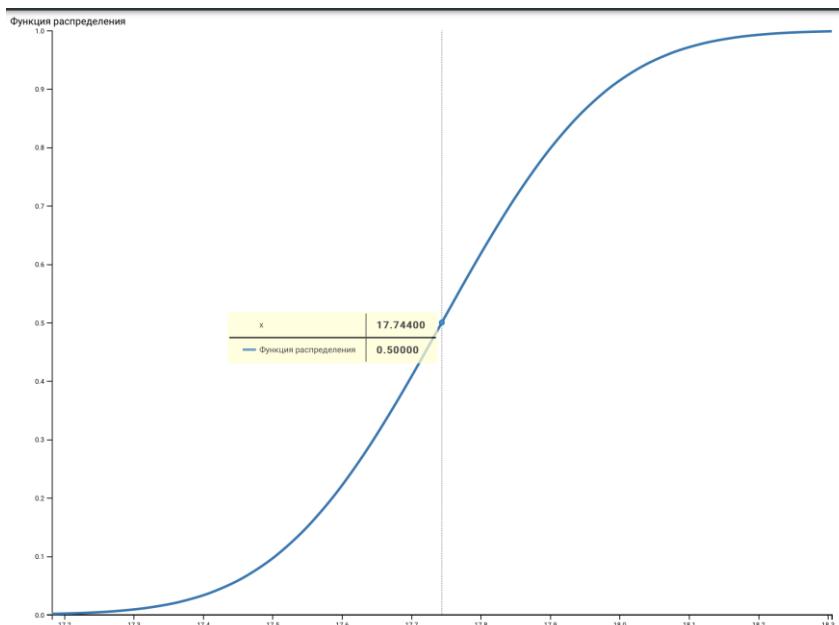


График плотности вероятности для flops:

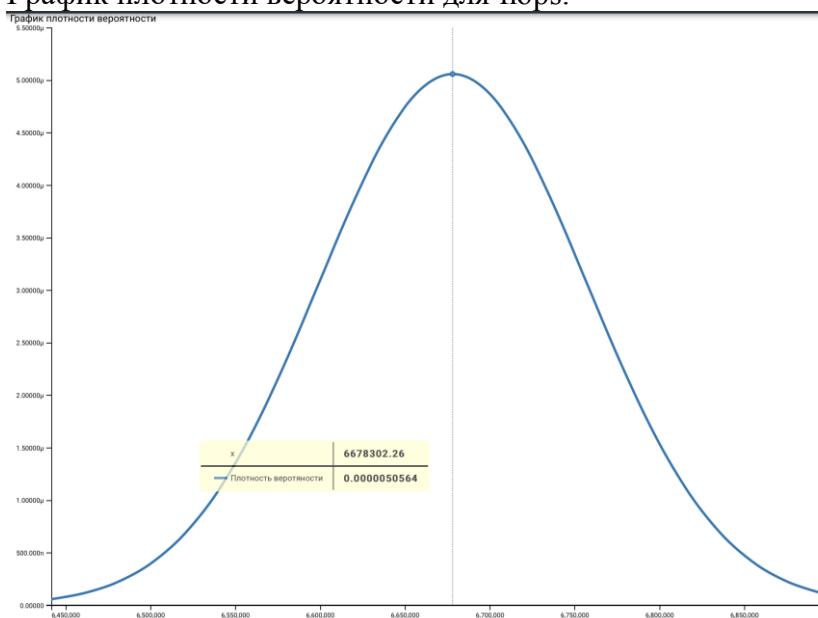
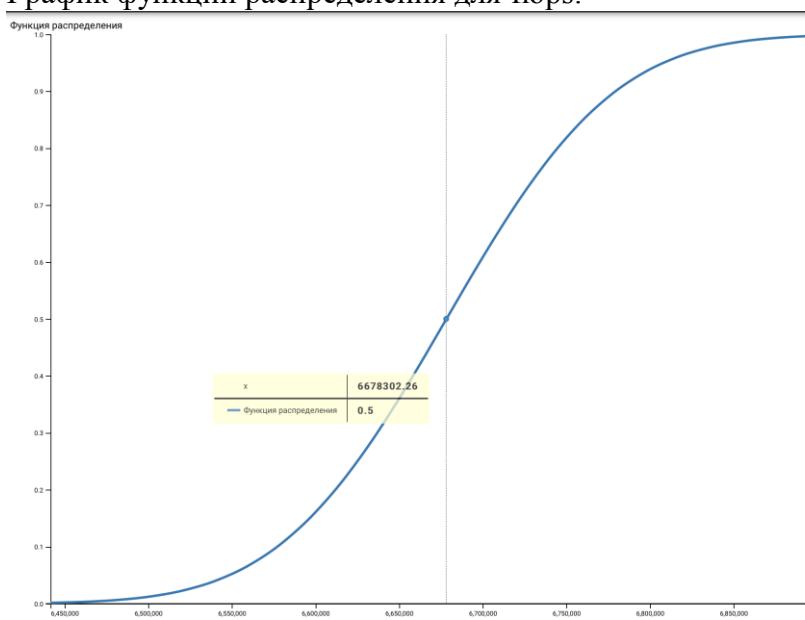


График функции распределения для flops:



**Для taskset -c 1 chrt -f nice -n 19**

Дисперсия D для time: 0,15473724

Дисперсия D для flops: 22433003888

Математическое ожидание m для time: 18,0164286

Математическое ожидание m для flops: 6580671,399

График плотности вероятности для time:

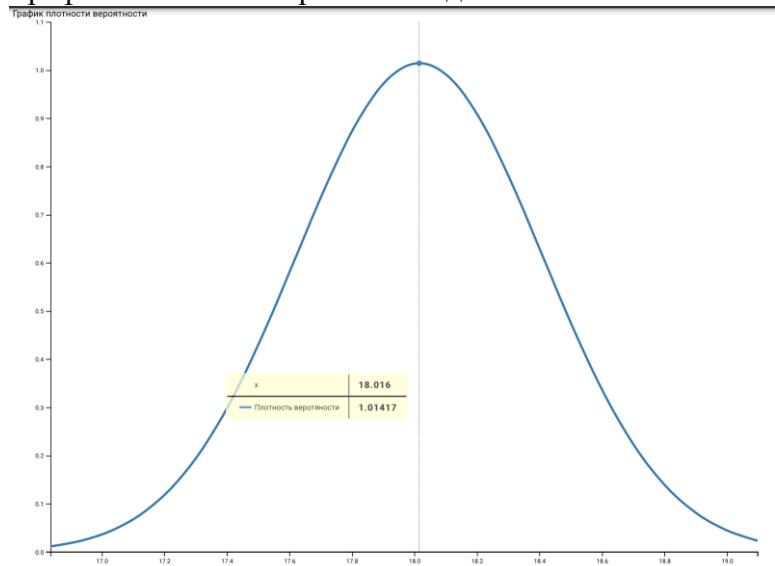


График функции распределения для time:

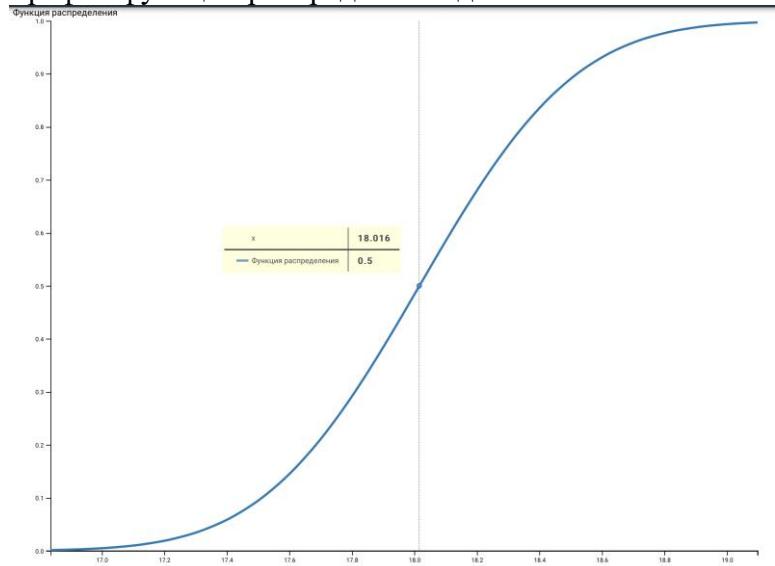


График плотности вероятности для flops:

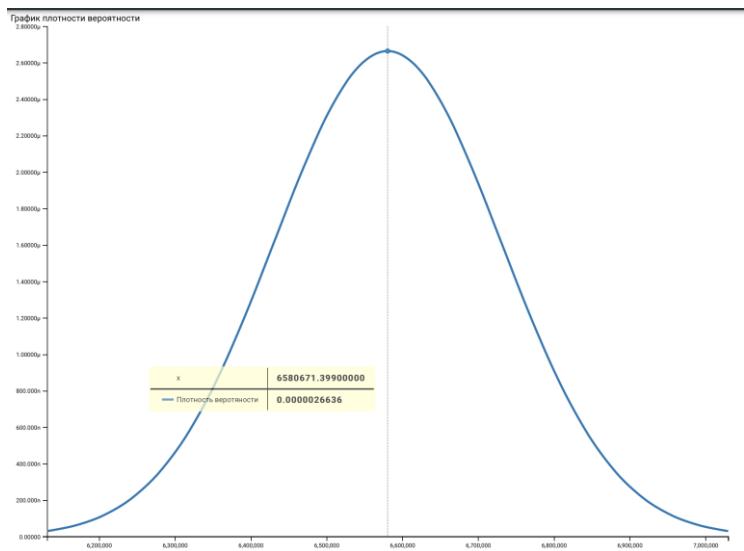


График функции распределения для flops:

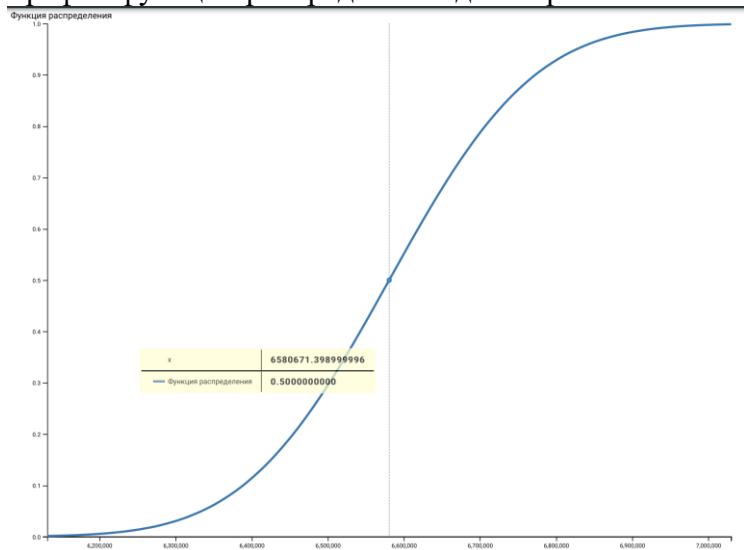


Таблица сравнения всех вариантов (в порядке возрастания):

Столбец1	$m(t)$	$D(t)$	$m(f)$	$D(f)$
nice -20	17,678	0,02100	6705244,655	3925920847
nice -5	17,718	0,04996	6693279,182	7810439252
nice 0	17,734	0,05990	6688874,329	9494506357
nice 1	17,735	0,01549	6680464,699	2921985828
nice 19	17,744	0,03508	6678302,260	6225004994
nice 5	17,796	0,09192	6662014,950	14526646082
cpu 1 nice 19	18,016	0,15474	6580671,399	22433003888

## Часть 2 (Kali OS)

Пересаживаемся на Kali Linux 2020, 3/6 ядер Intel Core I5 9600k, 4 Gib RAM.

Запретить выполнение всех потоков кроме того, который тестируется (путем запрета прерываний) или найти иной способ (найден)

Многозадачность Linux – вымешивающая (preemptive). Однако, нам надо, чтобы процессор (или конкретное ядро процессора) не делилось своим временем с неинтересными

процессами, а выполняло одну конкретную задачу до тех пор, пока та не завершится. Как же это сделать, не прибегая к перекомпиляции ядра с изменениями самого планировщика или конфигураций многозадачности. Есть 2 варианта:

1. Запретить выполнение всех потоков кроме того, который тестируется (путем запрета прерываний)

Идея неплохая, но для ее реализации необходимо изолировать отдельный нужный нам таск на каком-то ядре, убедиться, что на этом ядре он исполнится следующим или уже исполняется (зафорсить таск первым на ядро и/или сместить все другие потоки с ядра на другие ядра), после чего отключить прерывания (`local_irq_disable()` / `enable()` реализует `cli` / `sti`). Однако даже если это и сможет дать прирост в производительности для выбранного потока, уменьшить время его выполнения, - в целом данный метод является нерентабельным. Далее цитата из мануала на просторах веба:

**Duration** of the CPU state with disabled interrupts should be **short**, because it affects the whole OS. For that reason allowing *user space code* to be run with disabled interrupts is considered as **bad practice** and is not supported by the Linux kernel.

It is responsibility of the kernel module to wrap by `local_irq_disable` / `local_irq_enable` only the **kernel code**. Sometimes the kernel itself could "fix" incorrect usage of these functions, but that fact shouldn't be relied upon when write a module.

Из этого можно сделать вывод, что данный метод нерентабелен в условия задачи лабораторной работы: скорее всего запуск linpack'a из-под отключенных прерываний убьёт систему, т.к linpack сам по себе "тяжеловесен" по вычислениям и исполняется достаточно долго (он делает стресс тест путем вычислением корней большеразмерных систем линейных уравнений, запрет прерываний даже на одном ядре на такое продолжительное время скорее всего побьёт всю ось).

2. Можно на уровне ядра проэксплуатировать особенность алгоритма CFS планировщика процессов Linux.

Выбор следующего процесса осуществляется путем перестроения красно-чёрного дерева в соответствии с значениями `vruntime` его узлов (время, которое процесс уже «отработал» - следующим выбирается процесс, отработавший меньше всех). Можно просто из модуля ядра, пока процесс не завершится, писать в узел к/ч дерева, который соответствует процессу, значение `vruntime = 0`.

Давайте напишем модуль:

```
/* 
 * elev.c - The simplest kernel module.
 */
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <linux/kallsyms.h>
#include <linux/workqueue.h>
#include <linux/rcupdate.h>

static int myproc = 0;
static pid_t pidd;
module_param(myproc, int, 0);
static struct task_struct *tsk;
//static int res = -1;
//
void magic(struct work_struct *work);

DECLARE_WORK(my_dirty_work, magic);

void magic(struct work_struct *work){
    printk(KERN_INFO "elevator: starting work %d \n", tsk->pid);
    while (tsk != NULL){
        rCU_read_lock();
        tsk->se.vruntime = 0;
        rCU_read_unlock();
        tsk = pid_task(find_vpid(pidd), PIDTYPE_PID);
    }
    printk(KERN_INFO "elevator: bruh, your task has ended or never really existed");
}

static int __init elev_init(void)
{
    pidd = myproc;
    printk(KERN_INFO "elevator: got pid - %d \n", myproc);

    tsk = pid_task(find_vpid(pidd), PIDTYPE_PID);
    printk(KERN_INFO "elevator: got TASK by pid \n");
    if (tsk == NULL) goto ENDLI;
    schedule_work(&my_dirty_work);
ENDLI:
    printk(KERN_INFO "elevator: end of init");
    return 0;
}

static void __exit elev_exit(void)
{
    cancel_work_sync(&my_dirty_work);
    printk(KERN_INFO "elevator: goodbye\n");
}

module_init(elev_init);
module_exit(elev_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Lol_Lolinsky");
MODULE_DESCRIPTION("fuck preemptive scheduling");
```

Что мы здесь, по сути, делаем – запускаем поток ядра, kernel worker, который будет писать 0 в vruntime, пока процесс по pidу существует. Конечно, есть

небольшая вероятность, что система резко по завершению целевого процесса назначит его pid какому-нибудь другому и воркер будет работать вечно.

Теперь можно модуль протестировать. Напишем скрипт:

```
GNU nano 5.2                                         test
#!/bin/bash

for ((i=1; i<30; i++))
do
    out=$(nice -n $1 ./linpack | grep 65536 & insmod elev.ko myproc=$!)
    sleep 2
    rmmod elev.ko
    sleep 2
    timee=`echo $out | awk '{print $2}'` 
    flops=`echo $out | awk '{print $6}'` 
    echo "$timee, $flops" >> testsys.txt
done
```

Будем запускать linpack совместно с модулем, аргументом в модуль будем передавать pid linpack'a. Соберем данные по flops и времени исполнения в txt.

Так же используем обычный скрипт для проверки (который уже был использован ранее):

```
GNU nano 5.2
#!/bin/bash

for ((i=1; i<30; i++))
do
    out=$(nice -n $1 ./linpack | grep 65536)
    timee=`echo $out | awk '{print $2}'` 
    flops=`echo $out | awk '{print $6}'` 
    echo "$timee, $flops" >> testsys.txt
done
```

Теперь же опишем результаты тестов:

- Тесты без нагрузки (на системе ничего не крутится кроме сессий терминалов и служебных потоков):

Как можно увидеть на скриншотах ниже, при noload (без нагрузки) условиях работы linpack'a вперед вырывается стандартный nice -20 (он же и показал самый быстрый результат в предыдущей части лабы), на втором месте после него по скорости идут тесты с использованием модуля и найс -20 (на самом деле, как будет видно потом, nice -20 ничего не дает при использовании модуля ), а заключает тройку тесты с привязкой к 1 ядру (видимо, без нагрузки тратится время на принуждение процесса к ядру).

время	flops	услови	время	flops	услови	Column1	flops	услови
14,64	8136080,737		14,8	8052388,778		14,87	7987368,041	
14,89	7998405,819		14,79	8050330,307		14,79	8031164,718	
14,64	8139542,672		14,65	8145053,257		14,54	8196329,681	
15,01	7945775,929		14,77	8060950,852		14,58	8176626,157	
14,8	8055579,398		14,7	8114566,698		14,59	8175448,933	
15,06	7887473,871		14,68	8116427,588		14,65	8130536,009	
14,89	7984323,294		14,7	8109770,958		14,5	8224565,136	
15,1	7881385,418		14,73	8086809,161		14,62	8137214,279	
14,8	8048267,85		14,82	8029764,647		14,52	8212805,608	
14,97	7941104,7		14,86	8033799,952		14,64	8130677,033	
15	7924819,844		14,97	7943499,57		14,58	8175955,432	
14,84	8032821,932		14,78	8068166,857		14,71	8089865,691	
14,92	7987440,344		14,7	8098243,179		14,61	8164314,404	
14,78	8048281,524	taskset 1 nice -20	14,76	8065344,965	module nice -20	14,66	8128421,979	noload
14,74	8085232,736	noload	14,91	7979324,375	noload	14,67	8132526,214	nice -20
14,94	7976803,226		14,7	8112287,667		14,69	8107987,62	
14,86	8024234,343		14,73	8100929,916		14,68	8124819,134	
14,83	8035946,837		14,82	8043150,47		14,76	8065200,417	
14,87	8018858,11		14,74	8089195,31		14,66	8133991,022	
15,01	7937252,251		14,77	8067128,391		14,68	8123090,756	
14,91	7960140,148		14,75	8081380,707		14,65	8126276,764	
14,75	8072663,694		14,77	8081012,83		14,54	8207092,235	
14,97	7964031,061		14,79	8043546,538		14,55	8187427,113	
14,84	8030345,682		14,58	8177860,935		14,6	8155937,197	
15,04	7919958,555		14,91	7986721,627		14,66	8119861,095	
14,73	8090281,645		14,82	8039756,383		14,55	8193926,169	
14,73	8083161,07		14,79	8060894,539		14,64	8142784,33	
14,83	8034309,85		14,88	8006124,793		14,58	8166447,147	
14,81	8052745,408		14,8	8054092,242		14,57	8182832,794	
мат ож			мат ож			мат ож		
14,8689655	8010250,619		14,7748276	8065466,327		14,6324138	8142465,28	
0,01452389			0,00686158			0,00686897		

- Самое интересное – тесты с нагрузкой:

В качестве нагрузки решено запустить несколько вкладок youtube в firefox. Под этой нагрузкой результаты: 1) найс 0; 2) модуль; 3) найс -20

время	flops		время	flops		время	flops	
15,27	7774875,343		15,32	7763661,863		15,04	7917657,956	
15,14	7863561,052		14,88	8012428,258		15,03	7925688,684	
15,1	7885619,748		15,03	7922358,119		14,88	8007580,751	
15,16	7853847,314		14,95	7977262,076		15,25	7809232,009	
15,12	7877238,365		14,9	8004054,314		15,14	7859095,761	
15,1	7893119,695		15,08	7898292,55		15,1	7876662,041	
15,26	7784730,577		15,02	7933879,79		15,08	7883651,172	
15,31	7782416,339		14,93	7981619,901		14,92	7978626,922	
15,24	7801077,047		15,1	7874663,485		14,93	7978502,44	
15,12	7875627,49		15,08	7901795,069		14,98	7941465,555	
15,2	7834686,705		14,86	8023675,651		15,01	7936767,196	
15,17	7835270,544		14,79	8066957,026		15,23	7791551,373	
15,21	7818239,977		15,25	7791197,269		15,04	7901644,53	
15,15	7852057,602		15,05	7910357,435		15,05	7920193,839	
15,07	7905513,867		15,19	7825943,624		15,03	7914281,244	
15,23	7809768,012		15,19	7843443,875		14,96	7963982,436	
15,14	7847008,737		15,08	7895696,963		15,14	7867402,8	
15,03	7918327,375		15,1	7890578,697		15,05	7921777,963	
15,03	7913614,597		14,82	8039592,642		15,06	7905902,746	
15,11	7884904,73		14,88	8013909,344		14,99	7951216,842	
15,03	7916339,651		15,04	7926179,367		15,24	7808094,518	
15,08	7895999,671		15,07	7889467,87		15,03	7911660,535	
15,02	7928891,317		15,03	7911660,535		15,23	7820803,908	
15,14	7854386,717		15,11	7869112,825		15,38	7738725,075	
15,12	7858972,236		15	7945199,352		15,45	7688875,756	
15,13	7859846,603		15,01	7929670,338		15,12	7877576,202	
15,04	7908489,76		15,06	7912068,102		15,1	7893166,766	
15,06	7904902,156		14,95	7971395,028		15,08	7902254,35	
15,15	7852467,958		14,99	7943344,634		15,05	7891956,944	
мат ож			мат ож			мат ож		
15,1355172	7861789,696		15,0262069	7926533,31		15,08	7887905,22	
0,00602562			0,01529581			15,0927586	7886629,069	

Невероятно, но под нагрузкой модуль показал наилучший средневероятностный результат, с отрывом в 0.07 секунды от nice -20 и 0.1 секунды от обычного запуска linpack параллельно с ресурсоёмкими нагрузками firefox и youtube.